Tejas Nitin Joshi
#102181822
tejas.joshi@colorado.edu

ELEC 5623 RTES

ASSIGNMENT 2

Understand message queues and difference between standard and heap message passing

1. Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization"

The paper provides solution to combat scheduling problems like uncontrolled priority inversion, it delves into the usage of semaphores to prevent such blocking. The paper initially describes the “basic priority inheritance protocol” which introduces more complexities in terms of blocking. Apart from the direct blocking, deadlock problems, a new problem of push through blocking is encountered. The paper suggests that a remedy to the above problems would be to use a more enhanced version of the protocol called the “priority ceiling protocol” which would reduce the worst case execution time and preventing deadlocks. The paper goes on to describe a sufficient condition for determining schedulability for the same with a set of supporting equations based off Liu and Layland Paper and the Lehoczky, Sha & Ding papers. Further it provides some insight on the implementation considerations.

The paper introduces the current problems which were not discussed in previous papers and points out that shared resources between tasks are commonplace and would affect scheduling tremendously. To lock a resource for a task the concept of semaphore is used and due to which a problem called blocking (a kind of priority inversion) is introduced. An example for the same is provided with notations.

Priority Inheritance Protocol

The above protocol is designed to battle the blocking problem (by a lower priority task to a higher one indefinitely). With a series of lemmas and proofs it grows up to a scenario where deadlocks may occur and also a chain of blocking may be created while scheduling the tasks. The paper moves on to a second improved protocol to eliminate the above problems.

Priority Ceiling Protocol (PCP)

The protocol is introduced with example scenarios for deadlocks and chain blocking and with help of detailed steps of scheduling execution, explains how the priority ceiling conditions are defined, so as to prevent the deadlock and chain blocking scenarios. A series of lemmas proofs and corollaries corroborate the protocol. A newer form of blocking is introduced due to the new set of rules i.e. ceiling blocking but is a necessary form of blocking to avoid the other forms of blocking and worst case blocking is dramatically improved in whilst using the priority ceiling protocol.

The paper then goes on to discuss schedulability analysis for the PCP and develops an equation from the Liu & Layland paper for the RM analysis following eq. is a test considering the Blocking

$$\forall i,\ 1 \le i \le n, \qquad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Similarly a more specific version of the schedulability test similar to equation in the LSD paper along with compensation for the Blocking time Bi is given as:

$$\forall i,\ 1 \le i \le n,$$

$$\min_{(k,l)\in R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1$$

where C_i, T_i, and U_i are defined in Theorem 15, and B_i is the worst case blocking time for τ_i.

2. Heap_mq.c

The heap_mq.c file c introduces the concept of queue. The entry point function here is the heap_mq () function which fills up the imagebuffer array of 4096 elements with the ascii char 'A' as the starting point and incrementing the ascii value by 1 to 64, i.e. 0 to 63 and replacing the 64th character with the new line "\n" character for every iteration of the I loop which increments by 64, and runs 64 times to fill the array of 4096.

The last character of the array is set to null character to terminate the array properly. Again the 64th character of the array is set to null, hence appearing to be an array of 64 to all sting related functions.

Parameters for the que's maximum message and maximum msg _size is set.

Sizeof(void *) = 4bytes and sizeof(int) = 4 bytes assigns the buffer size to the queue. And the flag is set to 0,

A queue is opened with the right arguments with a creation and Read/Write flag and a name and is supplied the attributes via a struct whose specific were defined before.

The receiver and sender tasks are spawned with receiver having the higher priority.

The sender allocates a buffer for the data and copies the data from the image buffer to the malloc'ed memory and prints out the characters to be sent as shown int the terminal display screenshot below.

Then it prints the size of the buffer pointer, Two memcpy() functions copy the buffptr address and the id to the buffer and then send it to the queue with a priority of 30 with mq_send() function;

The sender goes into a taskDelay(3000);

The receiver task reads the queue using the mq_receive function and stores the pointer int queue to its local buffer and stores the priority locally.

Using two memcpy() functions again the pointer and the id is separated and printed out to the screen.
The next print uses the pointer and prints out the 64 values now available to the receiver task by passing a pointer through the queue.

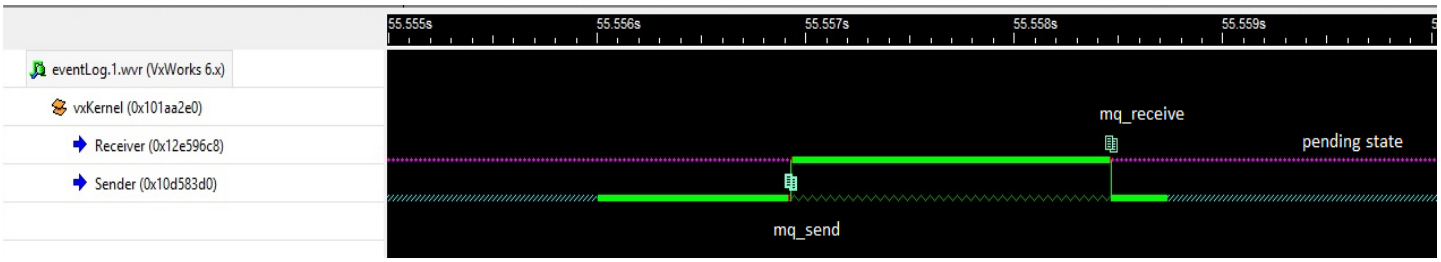
Finally the allocated space from the heap is destroyed.

An additional function to shut down the queues and tasks is also implemented.

```
-> heap_mq
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Receiver task spawned
Sender task spawned
value = 20 = 0x14
-> Reading 4 bytes
The size of test array is :64
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
receive: ptr msg 0x10D5BBF0 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
```

3. System Viewer analysis of heap_mq.c

The system viewer output reveals the message passing as shown in the screenshot below.



As viewed above the sender sends the message and the receiver receives it and goes into pending state as being executed in wile loop keeps checking the queue for new messages.

4. Posix_mq.c

The posix_mq.c has an entry point function called mq_demo() which sets up the attributes for the queue and spawns the sender and receiver tasks with the same priority as previously used in heap_mq.c.

Since sender has the higher priority it executes first and sets up /creates the queue. Since this is a one shot queue send and does not operate on sending messages in a while loop , creating a queue in the task suffices the purpose.

The receive task creates the queue and check contents of the queue and finds it to be empty and goes into the pending state.

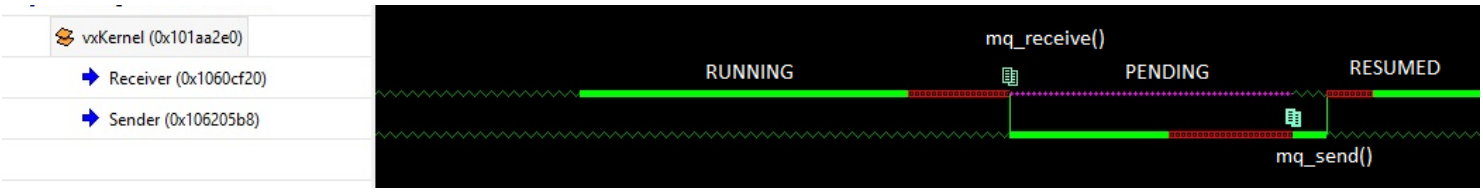
The sender pre-empts the receiver and mq_send()'s the canned message directly onto the queue. The receiver catches the queue contains something during the successive ticks and prints out the canned message which was passed along the queue along with the priority and the length as shown below in the terminal screenshot.

```
-> mq_demo
Receiver task spawned
Sender task spawned
value = 20 = 0x14
-> receive: msg this is a test, and only a test, in the event of
 a real emergency, you would be instructed ... received with pri
ority = 30, length = 95
send: message successfully sent

```

5. System Viewer Analysis of poxix_mq.c

The System viewer output is as shown in the screenshot below:



As seen above the receiver task runs first and creates the queue and performs an mq_receive() and goes into a pending state where it is pre-empted by the sender() task which sends the canned message using the function mq_send() and exits and the receiver resumes and performs the printing tasks and exits as well.